NASA-CR-203435

[illegible]

RNR-88-013

7N-61-CR
013548

IMPLEMENTATION AND ANALYSIS OF A NAVIER-STOKES ALGORITHM ON PARALLEL COMPUTERS

Raad A. Fatoohi
Sterling Software, Inc.
Palo Alto, CA 94303

Chester E. Grosch
Old Dominion University
Norfolk, VA 23529

Abstract -- This paper presents the results of the implementation of a Navier-Stokes algorithm on three parallel/vector computers. The object of this research is to determine how well, or poorly, a single numerical algorithm would map onto three different architectures. The algorithm is a compact difference scheme for the solution of the incompressible, two-dimensional, time dependent Navier-Stokes equations. The computers were chosen so as to encompass a variety of architectures. They are: the MPP, an SIMD machine with 16K bit serial processors; Flex/32, an MIMD machine with 20 processors; and Cray/2. The implementation of the algorithm is discussed in relation to these architectures and measures of the performance on each machine are given. Simple performance models are used to describe the performance. These models highlight the bottlenecks and limiting factors for this algorithm on these architectures. Finally conclusions are presented.

I. Introduction

Over the past few years a significant number of parallel computers have been built. Some of these have been one of a kind research engines, others are offered commercially. Both SIMD and MIMD architectures are included. A major problem now facing the computing community is to understand how to use these various machines most effectively. Theoretical studies of this question are valuable. However, we believe that comparative studies, wherein the same algorithm is implemented on a number of different architectures, provide an equally valid way to this understanding. These studies, carried out for a wide variety of algorithms and architectures, can highlight those features of the architectures and algorithms which make them suitable for high performance parallel processing. They can exhibit the detailed features of an architecture and/or algorithm which can be bottlenecks and which may be overlooked in theoretical studies. The success of this approach depends on choosing "significant" algorithms for implementation and carrying out the implementation over a wide spectrum of architectures. If the algorithm is trivial or embarrassingly parallel it will fit any architecture very well. We need to use algorithms which solve hard problems which are attacked in the scientific and engineering community.

In this paper we present the results of the implementation of an algorithm for the numerical solution of the Navier-Stokes equations, a set of nonlinear partial differential equations. In detail, the algorithm is a compact difference scheme for the numerical solution of the incompressible, two dimensional, time dependent Navier-Stokes equations. The implementation of the algorithm requires the setting of initial conditions, boundary conditions at each time step, time stepping the field, and checking for convergence at each time step. Equally important to the choice of algorithm is the choice of parallel computers. We have chosen to work on a set of machines which encompass a variety of architectures. They are: the MPP, an SIMD machine with 16K bit serial processors; Flex/32, an MIMD machine with 20 processors; and Cray/2. The basic comparison which we make is among SIMD instruction parallelism on the MPP, MIMD process parallelism on the Flex/32, and vectorization of a serial code on the Cray/2. The implementation is discussed in relation to these architectures and measures of the performance of the algorithm on each machine are given. In order to understand the performances on the various machines simple performance models are developed to describe how this algorithm, and others, behave on these computers. These models highlight the bottlenecks and limiting factors for algorithms of this class on these architectures. In the last section of this paper we present a number of conclusions.

II. The numerical algorithm

The Navier-Stokes equations for the two-dimensional, time dependent flow of a viscous incompressible fluid may be written, in dimensionless variables, as:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0, \tag{2.1}$$

$$\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} = \zeta, \tag{2.2}$$

$$\frac{\partial \zeta}{\partial t} + \frac{\partial}{\partial x}(u\,\zeta) + \frac{\partial}{\partial y}(v\,\zeta) = \frac{1}{\mathrm{Re}}\nabla^2 \zeta, \tag{2.3}$$

where $\vec{u} = (u,v)$ is the velocity, ζ is the vorticity and Re is the Reynolds number.

The numerical algorithm used to solve equations (2.1) to (2.3) was first described by Gatski, et al. [6]. This algorithm is based on the compact differencing schemes which require the use of only the values of the dependent variables in and on the boundaries of a single computational cell. Grosch [8] adapted the Navier-Stokes code to ICL-DAP. Fatoohi and Grosch [3] solved equations (2.1) and (2.2), the Cauchy-Riemann equations, on parallel computers. The algorithm is briefly described here.

Consider equations (2.1) to (2.3) in the square domain $0 \le x \le 1$, $0 \le y \le 1$ with the boundary conditions $u = 1$ and $v = 0$ at $y = 1$ and $u = v = 0$ elsewhere. Subdivide the domain into rectangular cells. The center of a cell is at $(i+1/2, j+1/2)$. Apply the centered difference operator to equations (2.1) to (2.2), to get

$$\delta_x U_{i+1/2,j+1/2} + \delta_y V_{i+1/2,j+1/2} = 0, \qquad (2.4)$$

$$\delta_x V_{i+1/2,j+1/2} - \delta_y U_{i+1/2,j+1/2} = \zeta_{i+1/2,j+1/2}. \qquad (2.5)$$

The adaptation of this algorithm to different parallel architectures can be simplified by the introduction of box variables to represent $\vec{U}$. The box variables, $\vec{P}$, are defined at the corners of the cells so that the average of two adjacent $\vec{P}$'s is equal to the $\vec{U}$ on the included side. The set of difference equations and boundary conditions in terms of the box variables are solved using a cell relaxation scheme which is equivalent to an SOR method [6], [8].

The compact difference approximation to equation (2.3) results in an implicit set of equations which are solved by an ADI method [4]. This method consists of two half steps to advance the solution one full step in time. Let Δt be the full time step and apply finite difference operators to equation (2.3), to get

$$\beta^{(x)}_{i,j}\, \zeta^{n+1/2}_{i-1,j} - (1 + 2\alpha^{(x)}_j)\, \zeta^{n+1/2}_{i,j} + \gamma^{(x)}_{i,j}\, \zeta^{n+1/2}_{i+1,j} = F_{i,j}, \qquad (2.6)$$

$$\beta^{(y)}_{i,j}\, \zeta^{n+1}_{i,j-1} - (1 + 2\alpha^{(y)}_i)\, \zeta^{n+1}_{i,j} + \gamma^{(y)}_{i,j}\, \zeta^{n+1}_{i,j+1} = G_{i,j}, \qquad (2.7)$$

where

$$F_{i,j} = -\beta^{(y)}_{i,j}\, \zeta^{n}_{i,j-1} - (1 - 2\alpha^{(y)}_i)\, \zeta^{n}_{i,j} - \gamma^{(y)}_{i,j}\, \zeta^{n}_{i,j+1},$$

$$G_{i,j} = -\beta^{(x)}_{i,j}\, \zeta^{n+1/2}_{i-1,j} - (1 - 2\alpha^{(x)}_j)\, \zeta^{n+1/2}_{i,j} - \gamma^{(x)}_{i,j}\, \zeta^{n+1/2}_{i+1,j},$$

$$\alpha^{(x)}_j = \Delta t \,/\, 2(\Delta x)^2_j \text{ Re}, \qquad \alpha^{(y)}_i = \Delta t \,/\, 2(\Delta y)^2_i \text{ Re},$$

$$\beta^{(x)}_{i,j} = \alpha^{(x)}_j + \Delta t\, U_{i-1,j} / 4(\Delta x)_j, \quad \beta^{(y)}_{i,j} = \alpha^{(y)}_i + \Delta t\, V_{i,j-1} / 4(\Delta y)_i,$$

$$\gamma^{(x)}_{i,j} = \alpha^{(x)}_j - \Delta t\, U_{i+1,j} / 4(\Delta x)_j, \quad \gamma^{(y)}_{i,j} = \alpha^{(y)}_i - \Delta t\, V_{i,j+1} / 4(\Delta y)_i.$$

The velocity field is not defined at the corners of the cells in this scheme; however, it can be computed using the box variables at the two immediate interior neighbors along the vertical and horizontal lines. Equation (2.6) represents a set of independent tridiagonal systems (one for each vertical line of the domain). Similarly, equation (2.7) represents a set of independent tridiagonal systems (one for each horizontal line of the domain). The ADI method for equation (2.3) is applied to all interior points of the domain. The values of ζ on the boundaries are computed using equation (2.2), see [2] for details.

The key to the adaptation of the relaxation scheme for solving equations (2.1) and (2.2) to parallel computers is the realization that each $\vec{P}$ is updated four times in a sequential sweep over the array of cells. This fact is utilized by using reordering to achieve parallelism. The computational cells are divided into four sets of disjoint cells so that the cells of each set can be processed in parallel [3]. It is therefore clear that the cell iteration for the box variables is a four "color" scheme. Thus four steps are necessary for a complete relaxation sweep.

The main issue in implementing the ADI method for equation (2.3) on parallel computers is choosing an efficient algorithm for the solution of tridiagonal systems. Two algorithms are considered here: Gaussian elimination and cyclic elimination, [4], [9]. The Gaussian elimination algorithm is based on an LU decomposition of the tridiagonal matrix. This algorithm is inherently serial because of the recurrence relations in both stages of the algorithm. However, if one is faced with solving a *set* of independent tridiagonal systems, then Gaussian elimination will be the best algorithm to use on a parallel computer; all systems of the set are solved in parallel. The cyclic elimination algorithm is a variant of the cyclic reduction algorithm [9] applying the reduction procedure to all of the equations and eliminating the back substitution phase of the algorithm. Cyclic elimination is most suitable for machines with a large natural parallelism, like the MPP.

The solution procedure for the Navier-Stokes equations can be summerized as follows:

(1) Assume that ζ is zero everywhere at $t = 0$. The variables and boundary values are initialized.

(2) The vorticity at the corners of the domain, undefined in this scheme, is approximated using the values of its neighboring points. The values of $\zeta_{i+1/2,j+1/2}$ are computed using the values of ζ at the corners of the cells.

(3) The relaxation process is implemented for each "color", i.e. four times in order to complete a sweep. The maximum residual is computed and tested against the convergence tolerance. The whole process is repeated until the iteration converges.

(4) The coefficients $\alpha^{(x)}_j$, $\alpha^{(y)}_i$, $\beta^{(x)}_{i,j}$, $\beta^{(y)}_{i,j}$, $\gamma^{(x)}_{i,j}$, $\gamma^{(y)}_{i,j}$ for both passes of the ADI method are computed.

(5) The values of ζ on the boundaries are computed.

(6) The tridiagonal equations distributed over columns, equation (2.6), are solved.

(7) The tridiagonal equations distributed over rows, equation (2.7), are solved.

These steps were implemented using the following subprograms: *setbc*, step (1); *zcntr*, step (2); *relaxd*, step (3); *cof*, step (4); *zbc*, step (5); *triied*, step (6); and *trijed*, step (7). The repetition of steps (2) through (7) yields the values of the velocity and vorticity at any later time.

III. Implementation on the MPP

The Massively Parallel Processor (MPP) is a large-scale SIMD processor built by Goodyear Aerospace Co. for NASA Goddard Space Flight Center [1]. The MPP is a back-end processor for a VAX-11/780 host, which supports its program development and I/O needs.

The MPP's high level language is MPP Pascal [7]. It is a machine-dependent language which has evolved from Parallel Pascal, an extended version of Pascal with a syntax for specifying array operations. These extensions provide a parallel array data type and operations on these arrays.

The Navier-Stokes algorithm, described in section II, was implemented on the MPP using 127 × 127 cells (128 × 128 grid points). The computational cells are mapped onto the array so that each corner of a cell corresponds to a processor. The seven subprograms of this algoritnm (see section I") were written in MPP Pascal.

These subprograms were executed entirely on the MPP; only I/O routines were run on the VAX.

The relaxation process, subprogram *relaxd*, was implemented on the array using the four color relaxation scheme [3]. The ADI method, subprograms *triied* and *trijed*, was implemented by solving two sets of 128 tridiagonal systems using the cyclic elimination algorithm. This is done in parallel on the array with a tridiagonal system of 128 equations being solved on each row or column.

One of the problems in solving Navier-Stokes equations on the MPP is the size of the PE memory. The relaxation subprogram uses almost all of the 1024 bit PE memory; 22 parallel arrays of floating point numbers, all but 5 of which are temporary. Although the staging memory can be used as a backup memory, this causes an I/O overhead and reduces the efficiency. This problem was solved by declaring all parallel arrays as global variables and using them in procedures for more than one purpose. Beside this memory problem, there are problems in using MPP Pascal to perform vector operations and to extract elements of parallel arrays. Operations on vectors are performed by expanding them to matrices and performing matrix operations; thus the processing rate is 1/128 of that for matrix operations. MPP Pascal does not permit extracting an element of a parallel array. This means that scalar operations involving elements of parallel arrays need to be expanded to matrix operations or performed on the VAX.

The relaxation subprogram is quite efficient; almost all of the operations are matrix operations, no vector and only two scalar operations per iteration, with data transfers only between nearest neighbors. The ADI subprograms are reasonably efficient; mostly matrix operations with few scalar and no vector operations. However, both algorithms have some hidden defects. In updating the box variables for each set in the relaxation scheme only one forth of the processors do useful work; the remaining processors are masked out. This is because only one corner of each cell of a set is updated each time. For each level of the elimination process in the cyclic elimination algorithm, a set of data is shifted off the array and an equal set of zeros is shifted onto the array. This means that some of the processors are not doing useful work; here they are either multiplying by zero or adding a zero. This is a problem with many algorithms on SIMD machines.

Table I contains the execution time for each subprogram of the algorithm, that for one iteration in the case of *relaxd*; the percentage of the total time spent in that subprogram; and the processing rate. It is clear, from Table I, that the majority of the time was spent in *relaxd* for this particular run. This is because the average time step requires about 270 iterations and the total time spent in the other subprograms (*zcntr, cof, zbc, triied, trijed*) is only about the time to do two iterations of *relaxd*. The number of iterations in *relaxd* per time step depends on the data used during a given run. A different input data set could result in a smaller number of iterations per time step and relatively less time spent in the relaxation subprogram.

Table I. Measured execution time and processing rate of the Navier-Stokes subprograms for the 128 × 128 problem on the MPP.

Sub-program	Execution time (msec)	Perc. of time (%)	Processing rate (MFLOPS)
setbc	0.587	0.00	84
zcntr	2.694	0.06	24
relaxd	15.265*	99.23	156
cof	1.933	0.05	136
zbc	1.833	0.04	1.1
triied	12.717	0.31	125
trijed	12.725	0.31	125
overall#	41.597	100.00	155

* per iteration.

# for ten time steps (execution time is in seconds here).

The processing rates in Table I are determined by counting only the arithmetic operations which truly contribute to the solution. Scalar and vector operations which were implemented as matrix operations are counted as scalar and vector operations. This is the reason why the subprograms *zbc* and *zcntr* have low processing rates; *zbc* has only vector operations while *zcntr* has some scalar operations implemented as matrix operations. The subprogram *setbc* has mostly scalar and data assignment operations which reduce its processing rate. Beside these three subprograms, the processing rate ranges from 125 to 155 MFLOPS with an average rate of about 140 MFLOPS.

In order to estimate the execution time of an algorithm on the MPP, the numbers of arithmetic and data transfer operations are counted and the cost of each operation is measured. This is illustrated in the following model. Only operations on parallel arrays are considered.

The execution time of an algorithm on the MPP, T, can be modeled as:

$$T = T_{cmp} + T_{cmm}, \qquad (3.1)$$

$$T_{cmp} = t_c\,(N_a\,C_a + N_m\,C_m + N_d\,C_d), \qquad (3.2)$$

$$T_{cmm} = t_c\,(N_{sh}\,C_{sh} + N_{st}\,C_{st}), \qquad (3.3)$$

where T_{cmp} and T_{cmm} are the computation and communication times; t_c is the machine cycle time (t_c = 100 nsec); N_a, N_m, N_d, N_{sh}, and N_{st} are the numbers of additions, multiplications, divisions, shift operations, and steps shifted; and C_a, C_m, C_d, C_{sh}, and C_{st} are the numbers of cycles for addition, multiplication, division, startup shift operation, and each step of shift operation. Table II contains the measured values of the basic floating point operations.

Table II. Measured execution times (in machine cycles) of the floating point operations in MPP Pascal.

Add	Multiply	Divide	One step shift	k step shift
965	811	1225	168	$136 + 32\,k$

Table III contains the operation counts per grid point for the Navier-Stokes subprograms on the MPP using the cyclic elimination algorithm for solving the tridiagonal systems. Note that scalar and vector operations (in *zcntr* and *zbc)*, which were implemented as matrix operations, are considered here as matrix operations. Table IV contains the estimated computation and communication times using equations (3.2) and (3.3) and Tables II and III. The cost of scalar operations is not included in this model; this explains the differences between the estimated and measured times for *setbc* and *cof*. Apart from these two subprograms, the difference between the total estimated and measured times ranges between 3% to 8% of the measured times. The amount of time spent on data transfers is quite modest; from 6% for *relaxd* to 25% for *triied* and *trijed*. This is because this algorithm does not contain many data transfers and these transfers are only between nearest neighbors except for the tridiagonal solvers.

Table III. Operation counts per grid point for the Navier-Stokes subprograms on the MPP, using the cyclic elimination algorithm for solving the tridiagonal systems.

Sub-program	Add	Multiply	Divide	Shift	Steps shifted
setbc	1	1	1	-	-
zcntr	15	9	-	19	28
*relaxd**	119	26	-	42	84
cof	8	8	-	8	8
zbc	5	7	4	8	11
triied	30	45	22	44	764
trijed	30	45	22	44	764

* per iteration.

Table IV. Estimated execution times (in milliseconds) of the Navier-Stokes subprograms on the MPP.

Sub-program	Comp. time	Comm. time	Total est. time	Measured time
setbc	0.300	-	0.300	0.587
zcntr	2.177	0.348	2.525	2.694
relaxd	13.592	0.840	14.432	15.265
cof	1.421	0.134	1.555	1.933
zbc	1.540	0.144	1.684	1.833
triied	9.239	3.043	12.283	12.717
trijed	9.239	3.043	12.283	12.725

IV. Implementation on the Flex/32

The Flex/32 is an MIMD shared memory multiprocessor based on 32 bit National Semiconductor 32032 microprocessor and 32081 coprocessor [5]. The results presented here were obtained using the 20 processor machine at NASA Langley Research Center.

The machine has ten local buses; each connects two processors. These local buses are connected together and to the common memory by a common bus. The 2.25 Mbytes of the common memory is accessible to all processors. Each processor contains 4 Mbytes of local memory. Each processor has a cycle time of 100 nsec.

The Navier-Stokes algorithm, described in section II, was implemented on the Flex/32 using 64 × 64 grid points (63 × 63 cells) and 128 × 128 grid points (127 × 127 cells). The main program as well as the seven subprograms of the algorithm were written in Concurrent Fortran, which comprises the standard Fortran 77 language and extensions that support concurrent processing.

The parallel implementation of the Navier-Stokes algorithm is done by assigning a strip of the computational domain to a process and performing all the steps of the algorithm in each process. The main program performs only the input and output operations and creates and spawns the processes on specified processors. In our implementation, we used 1, 2, 4, 8, and 16 processors of the machine. The domain is decomposed first vertically for the first six subprograms (*setbc, zcntr, relaxd, cof, zbc,* and *triied*) and then horizontally for the subprogram *trijed*. The relaxation scheme for each strip was implemented locally. After relaxing each set of cells, each process exchanges the values of the interface points with its two neighbors through the common memory. The tridiagonal equations were solved using the Gaussian elimination algorithm for both passes of the ADI method. Data is stored in the common memory, in the local memory of each processor, or in both of them.

In order to satisfy data dependencies between segments of the code, a counter is used. This counter, which is a shared variable with a lock assigned to it, can be incremented by any process and be reset by only one process. It is implemented as a "barrier" where all processes pause when they reach it. A set of flags are also used for synchronization in the relaxation subprogram.

Table V contains the speedups and efficiencies as functions of the number of processors for the 64 × 64 and 128 × 128 problems. The measured execution times and processing rates using 16 processors are listed in Table VI. The efficiency of the algorithm ranges from about 94%, for the 64 × 64 problem using 16 processors, to about 99%, for the 128 × 128 problem using two processors.

Table V. Speedup and efficiency as functions of the number of processors, *p*, of the Navier-Stokes algorithm on the Flex/32.

p	64 × 64 points		128 × 128 points	
	speedup	efficiency	speedup	efficiency
1	1.000	1.000	1.000	1.000
2	1.959	0.980	1.976	0.988
4	3.893	0.973	3.941	0.985
8	7.715	0.964	7.850	0.981
16	15.027	0.939	15.483	0.968

Table VI. Measured execution times for ten time steps and processing rates for the Navier-Stokes algorithm using 16 processors of the Flex/32.

Problem size (grid points)	Execution time (sec)	Processing rate (MFLOPS)
64 × 64	268.7	1.09
128 × 128	2587.1	1.13

The performance model is based on estimating the values of the overheads resulting from running the algorithm on more than one processor. The execution time of an algorithm on p processors of the Flex/32, T_p, can be modeled as:

$$T_p = T_{cmp} + T_{ovr}, \tag{4.1}$$

where T_{cmp} is the computation time and T_{ovr} is the overhead time. Let f_{ld} be a load distribution factor where $f_{ld} = 1$ if the load is distributed evenly between the processors and $f_{ld} > 1$ if at least one processor has less work to do than the other processors. Then the computation time on p processors can be computed by

$$T_{cmp} = f_{ld}\, T_1 \,/\, p, \tag{4.2}$$

where T_1 is the computation time using a single processor.

The overhead time can be modeled by:

$$T_{ovr} = T_{cmo} + T_{spn} + T_{syn}, \tag{4.3}$$

where T_{cmo} is the common memory overhead time, T_{spn} is the spawning time of p processes, and T_{syn} is the synchronization time. Three components of the common memory overhead time can be identified:

$$T_{cmo} = T_{cma} + T_{cpl} + T_{cml}, \tag{4.4}$$

where T_{cma} is the common memory additional time - this results from storing additional variables in the common memory; T_{cpl} is the common plus local memory time - this results from storing variables in both the common and local memories; T_{cml} is the common minus local memory time - this results from storing variables in the common memory instead of local memory. The values of T_{spn}, T_{syn}, T_{cma}, T_{cpl}, and T_{cml} can be estimated as follows:

$$T_{spn} = p\, t_{spn}, \tag{4.5}$$

$$T_{syn} = p\, k_{lck}\, t_{lck}, \tag{4.6}$$

$$T_{cma} = n\, k_{cma} f_{bc}(p)\, t_{cma}, \tag{4.7}$$

$$T_{cpl} = n\, k_{cpl}\, (f_{bc}(p)\, t_{cma} + t_{lma}), \tag{4.8}$$

$$T_{cml} = n\, k_{cml}\, (f_{bc}(p)\, t_{cma} - t_{lma}), \tag{4.9}$$

where t_{spn} is the time to spawn one process - a reasonable value is 13 *m*sec; t_{lck} is the time to lock and unlock a variable - a reasonable value is 47 μsec; t_{cma} is the time to access a variable in common memory - a reasonable value is 6 μsec; t_{lma} is the time to access a variable in local memory - a reasonable value is 5 μsec; k_{lck} is the number of times a variable is locked and unlocked for each process; k_{cma} is the number of times an additional variable is referenced; k_{cpl} is the number of times a variable is stored in both local and common memory; k_{cml} is the number of times a variable is stored in common memory instead of local memory; and $f_{bc}(p)$ is the bus contention factor - it is a function of p. It is assumed that all memory operations are performed on vectors of length n.

The performance of the Navier-Stokes algorithm is heavily influenced by the performance of the relaxation subprogram; about 98% of the total time was spent in this subprogram. Since the number of cells is not divisible by the number of processors used, the last processor has less work to do than the other processors. Therefore, the load distribution factor, equation (4.2), can be computed by

$$f_{ld} = \left\lceil \frac{n-1}{p} \right\rceil \left(\frac{p}{n-1} \right). \tag{4.10}$$

Using the performance model, equations (4.1) through (4.10), the overhead time represents at most 5% of the execution time of the algorithm, including the load distribution factor. The overhead time of the relaxation subprogram dominates the total overhead time. The values of k_{lck} and k_{cma} for each iteration of the relaxation process are 1 and 8. The spawning time has a minor impact on the overhead time because the processes are spawned only once during the lifetime of the program. The synchronization time is insignificant because the routines that provide the locking mechanism are very efficient and overlap with the memory access. The bus contention factor is very small. The common memory additional time, T_{cma}, dominates the overhead time. This overhead results from accessing the interface points for each iteration of the relaxation subprogram. The other components of the common memory overhead time, T_{cpl} and T_{cml}, have a negligible impact on the total overhead time because these operations are performed only once during every time step.

V. Implementation on the Cray/2

The Cray/2 is an MIMD supercomputer with four Central Processing Units, a foreground processor which controls I/O and a main memory. The results reported here were obtained using the old Cray/2 at NASA Ames Research Center; the new one has a shorter main memory access time than the old one.

The Navier-Stokes algorithm, described in section II, was implemented on one processor of the Cray/2 using 64 × 64 and 128 × 128 grid points. The reordered form of the relaxation scheme, the four color scheme, was implemented on the Cray/2 with no major modifications. The reordering process removes any recursion because each of the four sets (colors) contains disjoint cells. The two sets of the tridiagonal systems were solved by the Gaussian elimination algorithm for all systems of each set in parallel. This was done by changing all variables of the algorithm into vectors running across the tridiagonal systems. The inner loops of all of the seven subprograms of the Navier-Stokes algorithm were fully vectorized. The local memory was used to store some of the variables, whenever that was possible. This reduces main memory conflicts and speeds up the calculation.

Tables VII and VIII contain the execution time for each subprogram, the percentage of the total time spent in that subprogram, and the processing rate for the 64 × 64 and 128 × 128 problems. Most of the time was spent in *relaxd*, and the average time step requires about 110 iterations for the 64 × 64 problem and about 270 iterations for the 128 × 128 problem. The subprogram *setbc* has a low processing rate because it has mostly memory access and scalar operations; however, this subprogram is called only once during the lifetime of the program. Beside this subprogram, the processing rate ranges from 57 to 97 MFLOPS with an average rate of about 70 MFLOPS for the subprograms of both problems.

Table VII. Measured execution time and processing rate of the Navier-Stokes subprograms for the 64 × 64 problem on one processor of the Cray/2.

Sub-program	Execution time (msec)	Perc. of time (%)	Processing rate (MFLOPS)
setbc	0.480	0.02	25
zcntr	0.252	0.08	63
relaxd	2.719*	99.02	96
cof	0.720	0.24	85
zbc	0.015	0.01	66
triied	1.007	0.33	57
trijed	0.928	0.30	62
overall#	3.048	100.00	96

* per iteration

# for ten time steps (execution time is in seconds here).

Table VIII. Measured execution time and processing rate of the Navier-Stokes subprograms for the 128 × 128 problem on one processor of the Cray/2.

Sub-program	Execution time (msec)	Perc. of time (%)	Processing rate (MFLOPS)
setbc	1.651	0.01	29
zcntr	1.059	0.03	61
relaxd	11.001*	99.60	97
cof	3.036	0.10	84
zbc	0.034	0.00	59
triied	4.014	0.13	59
trijed	3.870	0.13	62
overall#	30.286	100.00	97

* per iteration

# for ten time steps (execution time is in seconds here).

Based on the fact that Cray vector operations are "stripmined" in sections of 64 elements, the time required to perform arithmetic and memory access operations on vectors of length L_{vcr} can be modeled as follows:

$$T_{f1} = (\left\lceil \frac{L_{vcr}}{64} \right\rceil L_f + L_{vcr})\, N_{f1}\, CP, \qquad (5.1)$$

$$T_{f2} = (\left\lceil \frac{L_{vcr}}{128} \right\rceil L_f + \frac{L_{vcr}}{2})\, N_{f2}\, CP, \qquad (5.2)$$

$$T_{m1} = (\left\lceil \frac{L_{vcr}}{64} \right\rceil L_m + R_1 L_{vcr})\, N_{m1}\, CP, \qquad (5.3)$$

$$T_{m2} = (\left\lceil \frac{L_{vcr}}{128} \right\rceil L_m + R_2 \frac{L_{vcr}}{2})\, N_{m2}\, CP, \qquad (5.4)$$

where T_{f1} and T_{f2} are the times to perform floating point operations with strides of 1 and 2; T_{m1} and T_{m2} are the times to perform main memory access operations with strides of 1 and 2; CP is the clock period (CP = 4.1 nsec); L_m is the length of main memory to registers path (L_m = 56 CPs); L_f is the length of floating point functional unit (L_f = 23 CPs); R_1 and R_2 are the data transfer rates through main memory with strides of 1 and 2 (reasonable values are R_1 = 1 and R_2 = 3.5, although competition from other processors causes a lower transfer rates and hence increased values of R_1 and R_2); N_{f1} and N_{f2} are the numbers of floating point operations with strides of 1 and 2; and N_{m1} and N_{m2} are the numbers of main memory access operations with strides of 1 and 2.

Table IX contains the operation counts per grid point for the Navier-Stokes subprograms using the Gaussian elimination algorithm for solving the tridiagonal systems. These operations are performed on all grid points of the domain except for *zbc* where they are performed on vectors. Tables X and XI contain the estimated times of the Navier-Stokes subprograms for the 64 × 64 and 128 × 128 problems. These times are obtained using equations (5.1) to (5.4) and Table IX. It is assumed that each division takes four times the multiplication time. The main memory access time for each subprogram represents about 50% to 70% of the total estimated and measured time. This shows that the Cray/2 is a memory bandwidth bound machine. The memory stride of 2 in *relaxd* causes more than a 50% slowdown in data transfer rate. The difference between the total estimated and measured values can be attributed to several causes. Among these are: the memory access and arithmetic operations can overlap, the time to perform scalar operations is not included, and there is up to 20% offset on the results depending on the memory traffic and the number of the active processes. Finally, this model does not take into account the overlapping between segments of long vectors for the same operation. However, it was found that this overlapping is insignificant for Fortran programs.

Table IX. Operation counts per grid point for the Navier-Stokes subprograms on the Cray/2, using the Gaussian elimination algorithm for solving the tridiagonal systems.

Sub-program	Add	Multiply	Divide	Memory access
setbc	1	1	1	8
zcntr	3	1	-	5
*relaxd**	46	20	-	31
cof	8	8	-	16
zbc#	5	7	4	20
triied	6	7	2	17
trijed	6	7	2	17

* per iteration   # vector operations.

Table X. Estimated and measured execution times (in milliseconds) of the Navier-Stokes subprograms for the 64 × 64 problem on one processor of the Cray/2.

Sub-prog.	Mem. time	Add time	Mult. time	Est. time	Measured time
setbc	0.246	0.022	0.111	0.379	0.480
zcntr	0.154	0.067	0.022	0.243	0.252
relaxd	1.915	1.206	0.551	3.672	2.719
cof	0.480	0.173	0.173	0.826	0.720
zbc	0.010	0.002	0.008	0.020	0.015
triied	0.510	0.130	0.324	0.964	1.007
trijed	0.510	0.130	0.324	0.964	0.928

Table XI. Estimated and measured execution times (in milliseconds) of the Navier-Stokes subprograms for the 128 × 128 problem on one processor of the Cray/2.

Sub-prog.	Mem. time	Add time	Mult. time	Est. time	Measured time
setbc	0.996	0.090	0.450	1.536	1.651
zcntr	0.622	0.270	0.090	0.982	1.059
relaxd	6.826	4.144	1.802	12.772	11.001
cof	1.967	0.711	0.711	3.389	3.036
zbc	0.019	0.004	0.016	0.039	0.034
triied	2.090	0.533	1.333	3.956	4.014
trijed	2.090	0.533	1.333	3.956	3.870

VI. Comparisons and Concluding Remarks

There are a number of measures that one can use to compare the performance of these parallel computers using a particular algorithm. One is the processing rate and another is the execution time (see Tables I, VI, VII and VIII). However it must be borne in mind that both of these measures depend on the architectures of the computers, the overhead required to adapt the algorithm to the architecture, and the technology, that is, the intrinsic processing power of each of the computers.

If we consider a single problem, a ten time step run of the algorithm on a 128 × 128 grid, then the processing rate is a maximum for the MPP, 155 MFLOPS, compared to 97 MFLOPS for the Cray/2, and only 1.13 MFLOPS on 16 processors of the Flex/32. The low processing rate of the algorithm on the 16 processors of the Flex/32 is simply due to the fact that the National Semiconductor 32032 microprocessor and 32081 coprocessor are not very powerful. Although the algorithm has a higher performance rate on the MPP than on the Cray/2, it takes less time to solve the problem on the Cray/2 than on the MPP. This is due to the algorithm overhead involved in adapting the algorithm to the MPP. As shown in Tables III and IX, each iteration of the relaxation process has 145 arithmetic operations per grid point on the MPP compared to 66 operations per grid point on the Cray/2. Also, the cyclic elimination algorithm, used on the MPP, has 92 arithmetic operations per grid point while the Gaussian elimination algorithm, used on the Cray/2, has only 10 operations per grid point; not including computation of the forcing terms.

The implementation of the algorithm on the Flex/32 has the same number of arithmetic operations per grid point as on the Cray/2; there is only a reordering of the calculations and no additional arithmetic operations in the overhead. The algorithmic overhead for the Flex/32 version is the cost of exchanging the values of the interface points and setting the synchronization counters for the relaxation scheme and accessing the common memory for the ADI method. This means that the code on each processor is the serial code plus the overhead code. When the code is run on one processor, it is just the serial code with the overhead portion removed.

Another measure of performance is the number of machine cycles required to solve a problem. This measure reduces the impact of technology on the performance of the machine. For the 128 × 128 problem, for example, the ten time step run requires about 416 billion cycles on the MPP, 7387 billion cycles on the Cray/2, and 25871 billion cycles on 16 processors of the Flex/32. This means that the MPP outperformed the Cray/2, by a factor of 18, and the latter outperformed the Flex/32, by a factor of 3.5, in this measure. This also means that one processor of the Cray/2 outperformed 16 processors of the Flex/32 even if we assume that both machines have the same clock cycle. The problem with the Flex/32 is that, although each processor has a cycle time of 100 nsec, the memories (local and common) have access times of about 1 μsec.

One simple comparison between the MPP and Cray/2 is the time to perform a single arithmetic operation using the models developed in sections III and V. Using equation (5.1), the time to perform a single floating point operation (addition or multiplication) on an array of size 128 × 128 elements on the Cray/2, excluding the memory access cost, is 91.3 μsec. The time to perform the same operation on the MPP using MPP Pascal, see Table II, ranges from 81.1 μsec (for multiplication) to 96.5 μsec (for addition). This shows that the processing power of a single functional unit of the Cray/2 is comparable to the processing power of the 16384 processors of the MPP. However, much of the overhead is not included in this comparison: memory access cost on the Cray/2, data transfers on the MPP, and so on.

This experiment showed that by reordering the computations we were able to implement the relaxation scheme on three different architectures with no major modifications. Two different algorithms, Gaussian elimination and cyclic elimination, were used to solve the tridiagonal equations on the three architectures; the two algorithms were chosen to exploit the parallelism available on these architectures. The algorithm exploits multiple granularities of parallelism. The algorithm vectorized quite well on the Cray/2. A fine grained parallelism, involving sets of single arithmetic operations executed in parallel, is obtained on the MPP. Parallelism at higher level, large grained, is exploited on the Flex/32 by executing several program units in parallel.

The performance model on the MPP was fairly accurate on predicting the execution times of the algorithm. The performance model on the Flex/32 showed the impact of various overheads on the performance of the algorithm. The performance model on the Cray/2 was based on predicting the execution costs of separate operations. This model is used to identify the major costs of the algorithm and reproduced the measured results with an error of at most 35%.

The ease and difficulty in using a machine is always a matter of interest. The Cray/2 is relatively easy to use as a vector machine. Existing codes that were written for serial machines can always run on vector machines. Vectorizing the unvectorized inner loops will improve the performance of the code. Unlike parallel machines, vector machines do not have the problem of "either you get it or not". The Flex/32 is not hard to use, except for the unavailability of debugging tools which is a problem for many MIMD machines (a synchronization problem could cause a program to die). On the other hand, the MPP is not a user-friendly system. The size of the PE memory is almost always an issue. MPP Pascal does not permit vector operations on the array nor does it allow extraction of an element of a parallel array. The MCU has 64 Kbytes of program memory. This memory can take up to about 1500 lines of MPP Pascal code. This means that larger codes can not run on the MPP. Finally, input/output is somewhat clumsy on the MPP. However, other machines with architectures similar to the MPP may not have the same problems that the MPP does.

There is one further observation of interest. This algorithm can be implemented concurrently on four processors of the Cray/2 (multitasking). The code will be similar to the Flex/32 version except that most of the variables should be stored in the main memory. Adapting this algorithm to a local memory multiprocessor with a hypercube topology should be relatively easy. A high efficiency is predicted in this case because all data transfers are to nearest neighbors and their cost should be very small compared to the computation cost.

Acknowledgements

We would like to thank David Wildenhain of SAR, Tom Crockett of ICASE, and Chris Willard of NAS for their assistance. This research was supported in part by NASA Contract NAS1-18107 while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665. Also, research was supported in part by NASA Contract NAS2-11555 while the first author was an employee of Sterling Software under contract to the Numerical Aerodynamic Simulation Systems Division at NASA Ames Research Center, Moffett Field, CA 94035.

References

[1] Batcher, K. E., "Design of a Massively Parallel Processor," **IEEE Trans. Comput.**, Vol. C-29, 1980, pp. 836-840.

[2] Fatoohi, R. A., **Implementation and Performance Analysis of Numerical Algorithms on the MPP, Flex/32 and Cray/2**, Ph.D. dissertation, Old Dominion Univ., Norfolk, VA, 1987.

[3] Fatoohi, R. A. and Grosch, C. E., "Implementation of a Four Color Cell Relaxation Scheme on the MPP, Flex/32 and Cray/2," **Proc. 1987 Int. Conf. Par. Proc.**, pp. 424-426.

[4] Fatoohi, R. A. and Grosch, C. E., "Implementation of an ADI Method on Parallel Computers," **J. Scientific Computing**, Vol. 2, No. 2, 1987, pp. 175-193.

[5] Flexible Computer Co., **Flex/32 Multicomputer System Overview**, Publication No. 030-0000-002, 2nd ed., Dallas, TX, 1986.

[6] Gatski, T. B., Grosch, C. E., and Rose, M. E., "A Numerical Study of the Two-Dimensional Navier-Stokes Equations in Vorticity-Velocity Variables," **J. Comput. Phys.**, Vol. 48, No. 1, 1982, pp. 1-22.

[7] Goddard Space Flight Center, **MPP Pascal Programmer's Guide**, Greenbelt, MD, 1987.

[8] Grosch, C. E., "Adapting a Navier-Stokes code to the ICL-DAP," **SIAM J. Scientific & Statistical Computing**, Vol. 8, No. 1, 1987, pp. s96-s117.

[9] Hockney, R. W. and Jesshope, C. R., **Parallel Computers: Architecture, Programming and Algorithms**, Adam Hilger, Bristol, England, 1981.